

Exp 5

Aim: Deploying a Voting/Ballot Smart Contract

Theory:

1. Importance of ‘require’ Statements in Solidity

In Solidity, the require statement is used to validate conditions before executing critical parts of a smart contract. It acts as a safety checkpoint that ensures only legitimate inputs and authorized users can proceed with a function call. If the specified condition evaluates to false, the transaction is immediately reverted, and any changes made during execution are undone. This protects the contract from invalid operations and preserves blockchain integrity.

In a Voting (Ballot) smart contract, require statements can be applied to enforce rules such as:

- Verifying that a voter has the right to vote.
- Preventing a voter from casting multiple votes.
- Restricting certain functions (like granting voting rights) to the chairperson only.

Additionally, require allows developers to include descriptive error messages, which improve debugging and user interaction. Overall, it strengthens contract security, reliability, and correctness.

2. Key Solidity Concepts: mapping, storage, and memory mapping

A mapping in Solidity is a key-value data structure used to associate one type of data with another. Its syntax is:

mapping(keyType => valueType) For example:

```
mapping(address => Voter) public voters;
```

In this case, each Ethereum address is linked to a Voter struct containing relevant information such as voting weight, voting status, and selected proposal.

Mappings are highly efficient for data retrieval and are widely used in voting contracts. However, they do not maintain a length property and cannot be directly iterated over, making them efficient for lookups but limited for enumeration.

Storage

Storage refers to the permanent data area on the blockchain. State variables declared in a contract are stored in storage by default. Data stored here remains available across multiple transactions unless modified.

Because storage writes consume significant gas, developers must use it carefully. For example, voter details stored in a mapping remain permanently saved throughout the contract's lifecycle.

Memory

Memory is temporary storage used during function execution. Variables declared with the memory keyword exist only for the duration of the function call and are discarded afterward.

Memory is less expensive compared to storage and is suitable for temporary variables, calculations, or function parameters. For instance, temporary string handling or intermediate computations are typically done in memory.

Smart contract developers must carefully choose between storage and memory to optimize performance and reduce gas costs.

3. Why Use bytes32 Instead of string?

In earlier versions of the Ballot contract, proposal names were often stored as bytes32 instead of string.

- bytes32 is a fixed-size data type that stores exactly 32 bytes.
- It is gas-efficient, easier to compare, and simpler for the EVM to handle.
- However, it limits text length to 32 characters, reducing flexibility.

On the other hand:

- string is a dynamic data type that allows variable-length text.
- It improves readability and user-friendliness.
- However, it requires more complex storage handling and consumes more gas.

Therefore, bytes32 is preferred when efficiency and lower gas costs are priorities, while string is chosen when readability and flexibility are more important.

Code:

```
// SPDX-License-Identifier: GPL-3.0 pragma solidity ^0.8.0;

/*
 *      @title Ballot
 *      @dev Implements voting process along with vote delegation
 */
```

```

contract Ballot {

    struct Voter {
        uint256 weight;          // weight is accumulated by delegation
        bool voted;              // if true, that person already voted
        address delegate;        // person delegated to
        uint256 vote;             // index of the voted proposal
    }

    struct Proposal {
        string name;              // proposal name
        uint256 voteCount;        // number of accumulated votes
    }

    address public chairperson;
    mapping(address => Voter) public voters;
    Proposal[] public proposals;

    /**
     * @dev Create a new ballot to choose one of 'proposalNames'.
     */
    constructor(string[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        for (uint256 i = 0; i < proposalNames.length; i++) {
            proposals.push(
                Proposal({
                    name: proposalNames[i],
                    voteCount: 0
                })
            );
        }
    }

    /**
     * @dev Give 'voter' the right to vote. Only chairperson can call.
     */
    function giveRightToVote(address voter) external {
        require(msg.sender == chairperson, "Only chairperson can give right to vote");
        require(!voters[voter].voted, "The voter already voted");
        require(voters[voter].weight == 0, "Voter already has voting rights");

        voters[voter].weight = 1
    }

    /**
     * @dev Delegate your vote to another voter.
     */
    function delegate(address to) external {
        Voter storage sender = voters[msg.sender];
        require(sender.weight > 0, "You have no right to vote");
        require(!sender.voted, "You already voted");
        require(to != msg.sender, "Self-delegation is not allowed");
    }
}

```

```

// Follow the chain of delegation
while (voters[to].delegate != address(0)) { to = voters[to].delegate;
require(to != msg.sender, "Delegation loop detected");
}

Voter storage delegate_ = voters[to];
require(delegate_.weight > 0, "Delegate has no right to vote");

sender.voted = true; sender.delegate = to;

if (delegate_.voted) {
// If the delegate already voted, add directly proposals[delegate_.vote].voteCount += sender.weight;
} else {
// If the delegate did not vote yet, add weight delegate_.weight += sender.weight;
}
}

/***
 * @dev Cast your vote.
*/
function vote(uint256 proposal) external { Voter storage sender = voters[msg.sender];
require(sender.weight > 0, "No right to vote"); require(!sender.voted, "Already voted");
require(proposal < proposals.length, "Invalid proposal index"); sender.voted = true

sender.vote = proposal; proposals[proposal].voteCount += sender.weight;

}

/***
 * @dev Returns index of winning proposal.
*/
function winningProposal() public view returns (uint256 winningProposal_) { uint256
winningVoteCount = 0;

for (uint256 p = 0; p < proposals.length; p++) {

if (proposals[p].voteCount > winningVoteCount) { winningVoteCount = proposals[p].voteCount;
winningProposal_ = p;
}

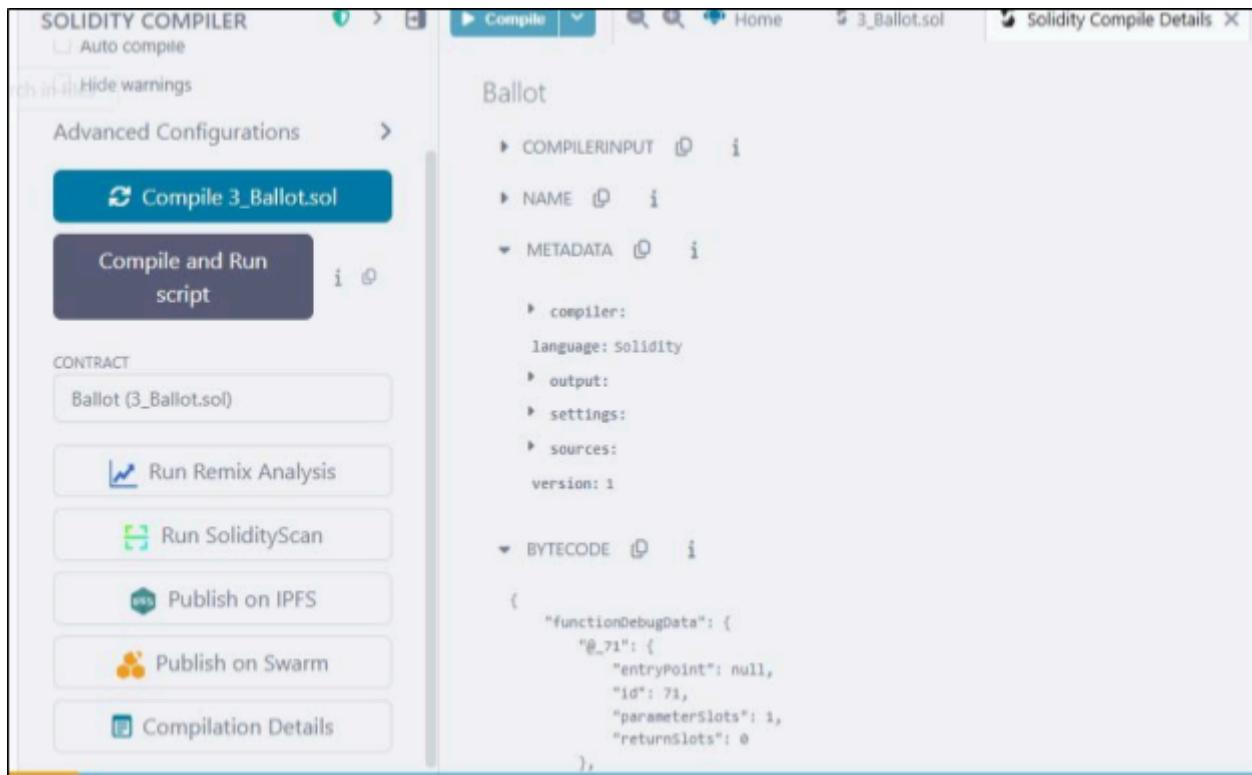
}
}
}

```

```
/**  
 * @dev Returns name of winning proposal.  
 */  
  
function winnerName() external view returns (string memory winnerName_) { winnerName_ =  
proposals[winningProposal()].name;  
}  
}
```

Output:

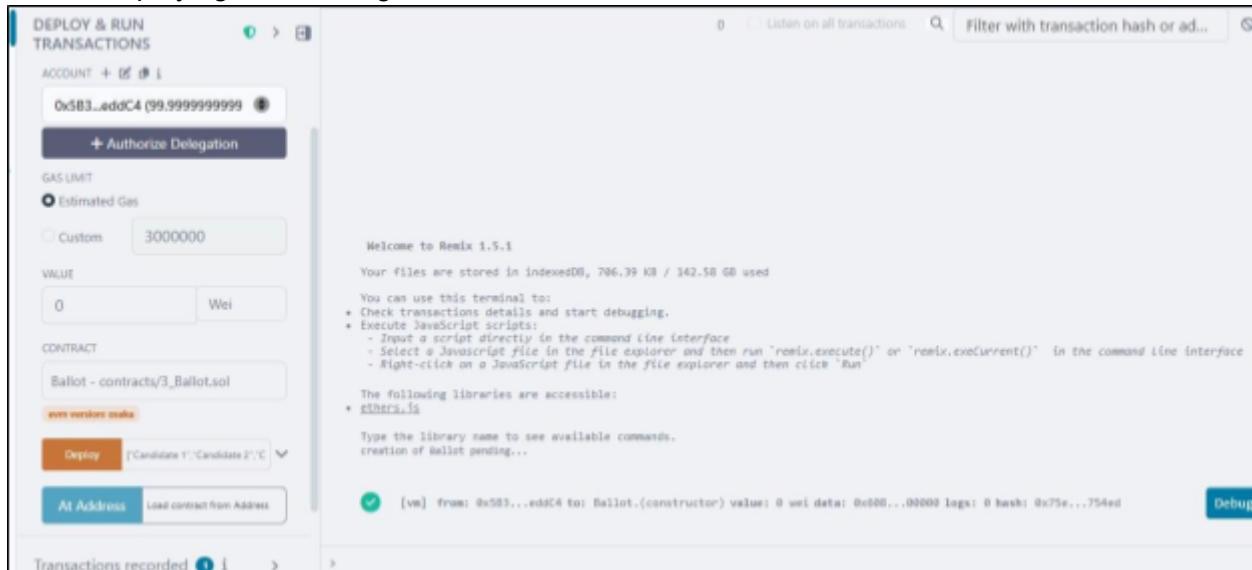
- Compiled Ballot.sol Contract



The screenshot shows the Solidity Compiler interface. On the left, there's a sidebar with buttons for 'Auto compile', 'Hide warnings', 'Advanced Configurations', 'Compile 3_Ballot.sol' (which is highlighted in blue), 'Compile and Run script', 'CONTRACT' (with 'Ballot (3_Ballot.sol)' selected), 'Run Remix Analysis', 'Run SolidityScan', 'Publish on IPFS', 'Publish on Swarm', and 'Compilation Details'. The main panel on the right displays the 'Ballot' contract details. It includes sections for 'COMPILERINPUT', 'NAME', 'METADATA' (with compiler, language, output, settings, sources, and version fields), and 'BYTECODE' (with a JSON object containing functionDebugData). The JSON code is as follows:

```
{
  "functionDebugData": {
    "@.71": {
      "entryPoint": null,
      "id": 71,
      "parametersSlots": 1,
      "returnSlots": 0
    }
  }
}
```

- Deploying and running of the contract



The screenshot shows the Remix interface. On the left, there are tabs for 'DEPLOY & RUN' (selected), 'TRANSACTIONS', and 'ACCOUNT' (with address 0x5B3...eddC4). Below these are buttons for '+ Authorize Delegation', 'GAS LIMIT' (set to 'Estimated Gas'), 'Custom' (gas limit 3000000), 'VALUE' (0 Wei), and 'CONTRACT' (selected 'Ballot - contracts/3_Ballot.sol'). There's also a dropdown for 'Deploy' with options like 'Candidate 1', 'Candidate 2', etc., and a button for 'At Address'. At the bottom left is a 'Transactions recorded' section. The right side of the interface has a terminal window with the following text:

Welcome to Remix 1.5.1
Your files are stored in indexedDB, 796.39 KB / 142.58 GB used
You can use this terminal to:
* Check transactions details and start debugging.
* Execute JavaScript scripts:
- Input a script directly in the command line interface
- Select a Javascript file in the file explorer and then run 'remix.execute()' or 'remix.executeCurrent()' in the command line interface
- Right-click on a JavaScript file in the file explorer and then click 'Run'
The following libraries are accessible:
* ethers.js
Type the library name to see available commands.
creation of Ballot pending...

At the bottom right, there's a 'Debug' button.

- Loading the Proposal Candidate's Names (string)

DEPLOY & RUN TRANSACTIONS

Deployed Contracts 1

BALLOT AT 0xD91...39138 (ME)

Balance: 0 ETH

delegate address to [not required]

giveRightToVote 9b849Ae677dD3315835cb2

vote uint256 proposal

chairperson

proposals uint256

voters address

winnerName

winningProposal

Low level interactions

CALldata

Voted successfully:

DEPLOY & RUN TRANSACTIONS

BALLOT AT 0xD91...39138 (ME)

Balance: 0 ETH

delegate address to

giveRightToVote address voter

vote

chairperson

proposals uint256

voters address

winnerName

winningProposal

Low level interactions

CALldata

Transact

You can use this terminal to:

- Check transactions details and start debugging.
- Execute Javascript scripts:
 - Input a script directly in the command line interface
 - Select a Javascript file in the file explorer and then run "remix.execute()" or "remix.executeCurrent()" in the command line interface
 - Right-click on a Javascript file in the file explorer and then click "Run"

The following libraries are accessible:

- ether.js

Type the library name to see available commands.

creation of Ballot pending...

[vm] From: 0x5B3...eddC4 To: Ballot.(constructor) Value: 0 Wei Data: 0x608...00000 Logs: 0 Hash: 0x75e...754ed Debug

[vm] From: 0x5B3...eddC4 To: Ballot.(constructor) Value: 0 Wei Data: 0x608...00000 Logs: 0 Hash: 0xeb8...8F9f6 Debug

[vm] From: 0x5B3...eddC4 To: Ballot.giveRightToVote(address) Value: 0 Wei Data: 0x9e7...35cb2 Logs: 0 Hash: 0x07...70233 Debug

[vm] From: 0x5B3...eddC4 To: Ballot.vote(uint256) Value: 0 Wei Data: 0x012...00001 Logs: 0 Hash: 0x6b5...d0e3 Debug

DEPLOY & RUN TRANSACTIONS

Deployed Contracts 1

BALLOT AT 0xd91...39138 (ME)

Balance: 0 ETH

Low level interactions

[vm] from: 0x5B3...eddC4 to: Ballot.(constructor) value: 0 wei data: 0x600...00000 logs: 0 hash: 0x75e...754ed creation of Ballot pending...

[vm] from: 0x5B3...eddC4 to: Ballot.(constructor) value: 0 wei data: 0x600...00000 logs: 0 hash: 0xeb8...8f9f6 transact to Ballot.giveRightToVote pending ...

[vm] from: 0x5B3...eddC4 to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei data: 0xe7...35cb2 logs: 0 hash: 0x697...70239 transact to Ballot.vote pending ...

[vm] from: 0xAB8...35cb2 to: Ballot.vote(uint256) 0xd91...39138 value: 0 wei data: 0x012...00001 logs: 0 hash: 0xeb5...dabe3 transact to Ballot.vote pending ...

[vm] from: 0x4B2...C02db to: Ballot.vote(uint256) 0xd91...39138 value: 0 wei data: 0x012...00002 logs: 0 hash: 0x78d...5467f transact to Ballot.vote errored: Error occurred: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "No right to vote".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

Tried to Vote twice which gave an error:

Deployed Contracts 1

BALLOT AT 0xd91...39138 (ME)

Balance: 0 ETH

Low level interactions

[vm] from: 0x617...5E7F2 to: Ballot.giveRightToVote pending ...

[vm] from: 0x617...5E7F2 to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei data: 0xe7...5e7f2 logs: 0 hash: 0x4f2...36204 transact to Ballot.giveRightToVote errored: Error occurred: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "Only chairperson can give right to vote".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

transact to Ballot.giveRightToVote pending ...

[vm] from: 0x617...5E7F2 to: Ballot.giveRightToVote(address) 0xd91...39138 value: 0 wei data: 0xe7...5e7f2 logs: 0 hash: 0xc6c...5920a transact to Ballot.giveRightToVote errored: Error occurred: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "Only chairperson can give right to vote".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

call to Ballot.winnerName

[call] from: 0x617F2E2f072FD005503197092aC168:93465E772 to: Ballot.winnerName() data: 0x20...a53f0

Voted from another node:

The screenshot shows the Remix IDE interface with the following details:

- Deployed Contracts:**
 - BALLOT AT 0x0f1...39138 (M)**
- Balance:** 0 ETH
- Transactions:**
 - delegate**: address to
 - giveRightToVote**: 0x817F282F072F0905583197002aC168c91465E7F2
 - vote**: 2
 - chairperson**
 - proposals**: uint256
 - voters**: address
 - winnerName**
 - winningProposal**: string winnerName_, Candidate 2
 - winningProposal_**: uint256 winningProposal_ 1
- Logs:**
 - revert**: The transaction has been reverted to the initial state. Reason provided by the contract: "Only chairperson can give right to vote". If the transaction failed for not having enough gas, try increasing the gas limit gently.
 - giveRightToVote**: [vm] from: 0x817F282F072F0905583197002aC168c91465E7F2 to: Ballot.giveRightToVote(address) 0x0f1...39138 value: 0 wei data: 0x0e?...5e?f2 logs: 0 hash: 0xc5c...5920e
 - vote**: [vm] from: 0x817F282F072F0905583197002aC168c91465E7F2 to: Ballot.vote(uint256) 0x0f1...39138 value: 0 wei data: 0x0e?...5e?f2 logs: 0 hash: 0xc5c...5920e
 - chairperson**: call to Ballot.chairperson
 - proposals**: [call] from: 0x817F282F072F0905583197002aC168c91465E7F2 to: Ballot.proposals() data: 0xe2b...a53fd
 - voters**: call to Ballot.voters
 - winnerName**: [call] from: 0x817F282F072F0905583197002aC168c91465E7F2 to: Ballot.winnerName() data: 0x000...ff3bd
 - winningProposal**: [call] from: 0x817F282F072F0905583197002aC168c91465E7F2 to: Ballot.winningProposal() data: 0x000...ff3bd

Checked the Winner node and the Proposals:

The screenshot shows the Remix IDE interface with the following details:

- Deploy & Run Transactions**
- Balance:** 0 ETH
- Transactions:**
 - delegate**: address to
 - giveRightToVote**: 0x17F6AD0Ef982297579C2
 - vote**: 2
 - chairperson**
 - proposals**: 2
- Logs:**
 - 0: string: name Candidate 3
 - 1: uint256: voteCount 2
 - 0: string: winnerName_ Candidate 3

Conclusion

In this experiment, a Voting (Ballot) smart contract was implemented and deployed using Solidity in the Remix IDE environment. The practical use of 'require' statements demonstrated how smart contracts enforce validation rules and prevent unauthorized or incorrect actions. Core Solidity concepts such as mapping, storage, and memory were examined to understand how data is managed efficiently on the blockchain.